CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Elsevier - Publisher Connector

Theoretical Computer Science 87 (1991) 251-262
Elsevier

On optimal parallel computations for sequences of brackets

Krzysztof Diks and Wojciech Rytter*
Institute of Informatics, Warsaw University, ul. Banacha 2, 00-913 Warszawa, Poland

Communicated by G. Mirkowska
Received January 1988
Revised August 1989

Abstract

Diks, K., and W. Rytter, On optimal parallel computations for sequences of brackets, Theoretical Computer Science 87 (1991) 251-262.

We present an optimal parallel algorithm ($\log_2 n$ time, $n/\log_2 n$ processors) for computing the matching function for a sequence of brackets and for transforming sequences of brackets to trees on the parallel access machine without read and write conflicts (EREW PRAM). It gives also an optimal parallel transformation on EREW PRAM of texts of expressions to expression-trees. Previously an optimal parallel algorithm for this problem was known (Bar-On, Vishkin (1985)) on a stronger model of parallel computations (CREW PRAM), where read conflicts were essential. It is not clear presently how big the difference is between the power of CREW and EREW PRAMs. Our result implies optimal parallel algorithms on EREW PRAM for several other algorithmic problems which previously had optimal parallel algorithms only on a CREW PRAM: expression evaluation (Abrahamson et al. (1987); Brent (1974); Gibbons, Rytter (1986); Miller, Reif (1985)); recognition of input-driven languages (Gibbons, Rytter (1988)); transforming regular expressions to finite automata (Rytter (1987)) and parsing bracket languages (Rytter, Giancarlo (1987)). If the tree of the expression is given then the expression can be optimally evaluated on the EREW PRAM, see Cole, Vishkin (1988); Kosaraju, Delcher (1988). However optimal parallel transformation of expression to corresponding trees was previously known only on the CREW PRAM. The structure of our algorithm for computing the matching function is similar to that of Bar-On and Vishkin (1985). The matching function is computed in the preprocessing phase for a subset of $O(n/\log_2 n)$ brackets and later it guides the computation for all brackets. Our initial subset of brackets is a subset of that used in Bar-On, Vishkin (1985). It is small enough to eliminate read conflicts in the preprocessing phase, however it complicates other phases.

1. Introduction

By an optimal parallel algorithm we here mean an algorithm working in $\log_2 n$ time with $n/\log_2 n$ processors. Optimality of such an algorithm depends also on the

* The results were first presented at the workshop "Sequences", Positano 1988.

0304-3975/91/$03.50 © 1991—Elsevier Science Publishers B.V.

model of parallel computations: the weaker the model, the stronger the result will be. We consider optimality for problems related to a fundamental problem in parallel computations: expression evaluation on PRAMs (parallel random access machines). If the expression is given by its text then the first problem here is the computation of the expression-tree.

The main problem in transforming the expression to an expression-tree is the computation of the matching function for sequences of brackets. Other parts of the transformation can be taken from [2], there are no read conflicts there. Using the matching function the transformation of a sequence of brackets to a corresponding tree can be computed by an optimal parallel algorithm on EREW PRAM. There are several types of PRAMs, differing with respect to the access to the global memory. CREW PRAM is the one which allows read conflicts and forbids write conflicts (no two processors can write simultaneously into the same location). EREW PRAM does not allow read conflicts, it is the weakest model in the family of PRAMs.

In the paper we avoid technicalities of PRAMs and use only one type of parallel instruction,

for each y in S **do in parallel** action(y);

Execution of such a statement consists in performing action(y) for all y in parallel.

The input to our algorithm is a sequence x of n brackets—a vector of symbols. It can be easily verified whether it is a well-formed sequence of brackets by an optimal parallel algorithm on EREW PRAM, hence we assume that we deal with well-formed sequences.

The output of the algorithm is the matching function (stored in the table) MATCH. For a bracket on the position i, MATCH[i] is the position of the corresponding matching bracket. For example if $x = (( )( ))$ then MATCH[1] = 6, MATCH[2] = 3, MATCH[3] = 2, ..., MATCH[6] = 1.

The algorithm uses the following simple observation: If we make all reductions possible within a given sequence of brackets then the sequence obtained will be of the form $)^i(^j$. Such sequences are called reduced sequences. We partition the input vector of brackets into parts of length $\log_2 n$. A processor is assigned to each part and it computes the function MATCH for pairs of brackets matched within its part. The brackets for which MATCH is computed are disregarded (erased) and afterwards each part contains a reduced sequence of brackets. We obtain a compressed representation x' of x. For example if

$$x = (( )(\mid ( ))(\mid)( )(\mid)( ))$$

then

$$x' = ((\mid)(\mid)(\mid))$$

(the symbols $\mid$ partition x).

W.l.o.g. we can assume that x has a compressed form. We identify the bracket with its position in the text. A bracket segment is a subinterval of [1..n] consisting

of one type of brackets, we can have left or right segments depending on the type of brackets. Left segments are denoted by $(_i^j$ and right segments are denoted by $)_i^j$. If the type of brackets is known then the bracket segment can be denoted by $[i..j]$. If we use the notation of bracket segments then the representation of x has only $O(n/\log_2 n)$ length because each reduced form consists of at most two bracket segments. The first step is to reduce the number of processors from n to $n/\log_2 n$, instead of n individual brackets we deal only with $O(n/\log_2 n)$ segments. In the algorithm we will represent reduced sequences of bigger parts of x by pairs of lists, a list of left segments and a list of right segments. The algorithm starts with a partial computation of the matching function. It is computed initially only for (later defined) splitting brackets. These brackets are indicated in Fig. 1.

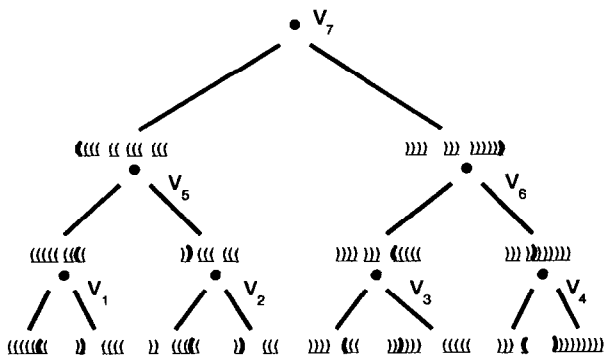

Fig. 1. The tree T with values at nodes representing lists of segments of brackets.

In each of $n/\log_2 n$ parts we distinguish two special brackets, the first left and the last right bracket. There are at most $2n/\log_2 n$ special brackets. The main idea of the algorithm of Bar-On and Vishkin [2] is to precompute MATCH for all special brackets, then the computation for all brackets is "driven" by the portion of MATCH computed in a preprocessing. The precomputation proceeds for each special bracket independently and takes $\log_2 n$ time. One processor deals with one bracket. There are no write conflicts but there are read conflicts. We show how to eliminate read conflicts for the Bar-On, Vishkin preprocessing phase by computing MATCH for a smaller subset of special brackets.

Assign $+1$ to each left and -1 to each right bracket. If the input sequence is $x_1x_2\ldots$ then define $\text{height}(i) = v(x_1)+\ldots+v(x_i)$, where $v(x)$ is the value assigned to the bracket x. The method is based on the following fact.

Fact 1.1. *If the ith position of a well-formed sequence contains a left bracket then* $\text{MATCH}(i)$ *is the first position* j *to the right of* i *with* $\text{height}(j) = \text{height}(i) - 1$.

The table height can be easily computed using a parallel prefix computation. We build a balanced tree T with $n/\log_2 n$ leaves, whose nodes correspond to parts of

the input sequence of brackets, see Fig. 1 and Fig. 2. Let height(w) be the minimum height of all leaves in the subtree rooted at node w. We show how the method works for left brackets, the case of right brackets is analogous.

Look at the example sequence in Fig. 1. There are 7 special left brackets, on positions 1, 10, 16, 23, 30, 39 and 47. We assign the ith processor to the special left bracket of the ith bracket part. The processor knows the position of its bracket, denote it by pos(i). Each processor will keep two additional informations: node(i), the node of T actually visited and phase(i), information about the phase of the algorithm which is performed by the processor. The structure of the algorithm is as follows.

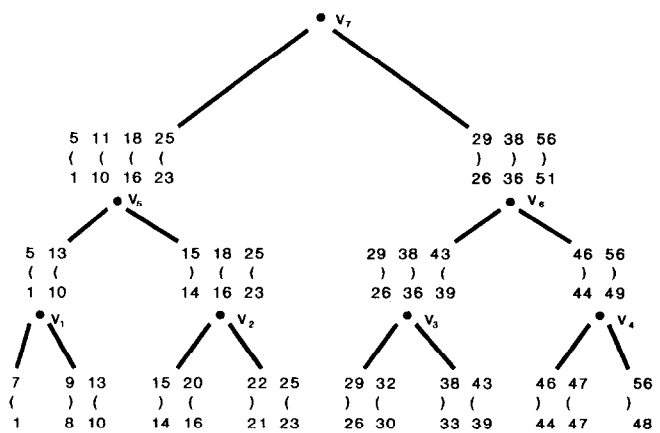

Fig. 2. The same tree as in Fig. 1, the segments of brackets are encoded as intervals.

Algorithm Partialmatch;
begin
  for each i **in parallel do** update(i);
    repeat $2\log_2 n$ times
      for each i **in parallel do**
        if phase(i) = *up* **then** phaseup(i) **else**
          if phase(i) = *down* **then** phasedown(i)
end.

Observe that if phase (i) is neither up nor down the processor i is idle. The procedures used in the algorithm are defined below.

procedure update(i);
begin
  if node(i) is a left son and
    height(brother(node(i))) < height(pos(i))
  then begin node(i) := brother(node(i)); phase(i) := down **end**
  else phase(i) := phase(node(i)) := up
end;

```
procedure phaseup(i);
begin
  node(i) := father(node(i)); update(i)
end;

procedure phasedown(i);
begin
  if node(i) is a leaf then
    begin
      MATCH(pos(i)) := the first bracket j in the segment corresponding to node(i)
      with height(j) = height(pos(i))) - 1; phase(i) = killed
    end
  else
    begin
      let left, right be the left, right son of node(i);
      if height(left) < height(pos(i)) then node(i) := left
      else node(i) := right
    end
end.
```

There are many read conflicts here because many processors can go up to the same node. To prevent such a situation we define another procedure.

```
procedure kill;
for each i in parallel do
  if node(i) is a right son and phase(node(i)) = phase(brother(node(i))) = up
  then phase(i) := killed
end.
```

We modify algorithm partialmatch, whenever two processors attempt to go up to the same father then the right one is killed.

```
Algorithm Partialmatch1;
begin
  for each i in parallel do update(i);
  repeat 2 log2 n times
  begin
    kill;
    for each i in parallel do
      if phase(i) = up then phaseup(i) else
        if phase(i) = down then phasedown(i)
  end
end.
```

The algorithm works in $\log_2 n$ time with $n/\log_2 n$ processors on EREW PRAM.

Let us look how the algorithm works on our example, see Fig. 1. After the first parallel updating (before the loop repeat) the vector phase is [up,up,up,up,down,up,down]. Then the first operation kill will set phase(2) = killed, phase(4) = killed. Processor 2 "looses" with 1 and 4 looses with 3. Then processors 1 and 3 are going up. The next kill sets phase(3) = killed. Therefore MATCH will be computed only for special left brackets in parts 1, 5, 6 and 7. There is no left bracket in the last part.

In the process of the computation of MATCH(i) for a special bracket i we traverse the tree T up, then in some moment we arrive at some node v, go to its brother w and go down. Let v_i be the (common) father of such v and w. The pair of brackets i, MATCH(i) is called the *splitting pair of* v_i and denoted by split(v_i). This pair has the property expressed as follows:

Fact 1.2. *Let v be an internal node of T and l, r be its left, right sons. If i is the leftmost left bracket in the leaves of the subtree rooted at l, and* MATCH(i) is in the leaves of the subtree rooted at r then split(v) = [i,MATCH(i)]. *The algorithm* Partialmatch1 *computes* MATCH *for all left special brackets occurring in splitting pairs.*

Analogously we can design an algorithm computing MATCH for special right brackets occurring in splitting pairs without read conflicts. In this way we are computing all splitting pairs. Observe that some nodes of T can have no splitting pair. We require that the leftmost left bracket in a subtree rooted at the left son is matched by a bracket in a subtree rooted at the right son of a given node. For some nodes such a situation can take no place.

The next section describes an algorithm computing MATCH for all brackets. The algorithm uses splitting pairs computed in out preprocessing.

2. The algorithm

Throughout the rest of the paper we use the example sequence x from Fig. 1 to demonstrate the algorithm.

$$x = (((((((\,|\,))((((\,|\,))((((((\,|\,))(((\,|\,))))(((\,|\,))))))(((((\,|\,)))(\,|\,)))))))).$$

The sequence is partitioned into 8 parts. Using bracket segments, x can be represented as follows:

$$\binom{7}{1}\,|\,)_{8}^{9}\,\binom{13}{10}\,|\,)_{14}^{15}\,\binom{20}{16}\,|\,)_{21}^{22}\,\binom{25}{23}\,|\,)_{26}^{29}\,\binom{32}{30}\,|\,)_{33}^{38}\,\binom{43}{39}\,|\,)_{44}^{46}\,\binom{47}{47}\,|\,)_{48}^{56}$$

We build a balanced binary tree T with $n/\log_2 n$ leaves corresponding to the parts of x, see Fig. 1 and Fig. 2. Let leaves(v) be the sequence of brackets corresponding to leaves of an internal node v of T. We define value(v) to be the reduced form

of leaves(v) represented as two lists of brackets, left and right lists, see Fig. 2. The operation compute(v) computes value(v) given values of the sons of v. In fact more important is a side effect of this operation—the computation of a pair of lists $(L_v, R_v) = \text{tobematched}(v)$. The structure of the whole algorithm looks as follows.

Algorithm COMPUTEMATCH;
begin
preprocessing:
  for each internal node v of T **do in parallel**
    compute the pair split(v) of splitting brackets of v;
    compute the function MATCH for splitting brackets;
compute-matching-lists:
  for $l = 1$ to $\log_2 n$ **do**
    for each node v at level l **do in parallel** compute(v);
    compute the matching lists Leftlist, Rightlist using values of tobematched;
makematch:
  using the matching lists Leftlist and Rightlist compute the vectors Leftvector, Rightvector;
  compute the function MATCH for all brackets
end. (of the algorithm).

We describe in detail each of the three phases.

2.1. The phase compute-matching-lists

First we describe by an example how the operation compute works. We show how the value of v_6 is computed, see Fig. 2. The computation concerns only brackets which are within leaves(v_6). The brackets in the segment [39..43] are matched with brackets in the segment [44..46] and a subsegment [49..50] of the segment [49..56]. The leftmost matched bracket is $(_{39}$ and the rightmost is $)_{50}$. This pair of brackets is the splitting pair of v_6. The splitting pair split(v_6) cuts from the sequence of lists

$$)_{26}^{29})_{36}^{38}(_{39}^{43})_{44}^{46})_{49}^{56}$$

the pair of sublists

$$L_{v_6} = (_{39}^{43} \quad \text{and} \quad R_{v_6} =)_{44}^{46})_{49}^{50}.$$

This pair is the value of tobematched(v_6). After cutting the lists from tobematched(v_6) we obtain value(v_6) as a pair of lists, a list of right segments and a list of left segments. In this case the second list is empty, see Fig. 2. The operation compute(v) is schematically presented in Fig. 3. The crucial point is to perform the operation compute in O(1) time for a given node. It can be easily done if we know the splitting pairs. We have to know also the current segments containing the splitting brackets. For each bracket which is not in any list in tobematched(v) for some v we keep

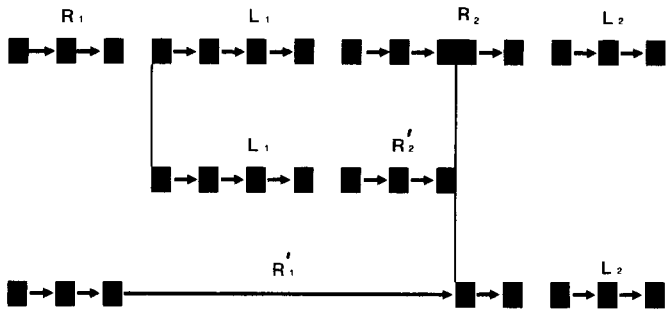

Fig. 3. The sons of v are v_1 and v_2. value$(v_1)=(R_1, L_1)$, value$(v_2)=(R_1, L_1)$, value$(v)=(R'_1, L_2)$. tobematched$(v)=(L_1, R'_2)$. In this case the weight of L_1 is less than that of R_2. weight(L_1) = weight(R'_2). The weight of the list is the total number of brackets it contains.

the name of the segment containing this bracket. This can be easily done at the beginning in the leaves of T.

After computing tobematched(v) for all internal nodes we concatenate all lists L_v in some order and obtain the list Leftlist. Then we concatenate lists reverse(R_v) in the same order and obtain the list Rightlist. The lists Leftlist and Rightlist are called matching lists. reverse(R) is the list obtained by listing the segments of R in the reverse order with each segment reversed (from higher positions to lower). Hence Leftlist $= L_{v_1}L_{v_2}\ldots$ and Rightlist $=$ reverse(R_{v_1})reverse$(R_{v_2})\ldots$

2.2. The phase makematch

Consider again the example $L_{v_1}=[6..7]$, $R_{v_1}=[8..9]$, $L_{v_2}[19..20]$, $R_{v_2}=[21..22]$, $L_{v_3}=[30..32]$, $R_{v_3}=[33..35]$, $L_{v_4}=[47..47]$, $R_{v_4}=[48..48]$, $L_{v_5}[12..13]$, $R_{v_5}=[14..15]$, $L_{v_6}=[39..43]$, $R_{v_6}=[44..46][49..50]$, $L_{v_7}=[1..5][10..11][16..18][23..25]$, $R_{v_7}=$ $[26..29][36..38][51..56]$. Hence

$$\begin{aligned}\text{Leftlist} = {} & [6..7][19..20][30..32][47..47][12..13][39..43][1..5]\\ & [10..11][16..18][23..25],\end{aligned}$$

$$\begin{aligned}\text{Rightlist} = {} & [9..8][22..21][35..33][48..48][15..14][50..49]\\ & [46..44][56..51][38..36][29..26].\end{aligned}$$

We compute the vectorial representation of Leftlist (Leftvector) and Rightlist (Rightvector). The brackets in the list are placed in the consecutive positions in the corresponding vector. Hence

$$\text{Leftvector} = [6, 7, 19, 20, 30, 31, 32, 47, 12, 13, 39, 40, 41, 42, \ldots]$$

and

$$\text{Rightvector} = [9, 8, 22, 21, 35, 34, 33, 38, 15, 14, 50, 49, 46, 45, \ldots]$$

The function MATCH is now easily computed. We set MATCH[6] = 9, MATCH[7] = 8, MATCH[19] = 22, MATCH[20] = 21, MATCH[30] = 35, MATCH[31] = 34, etc. It can be done in O(1) time with n processors or in $\log_2 n$ time with $n/\log_2 n$ processors.

The transformation of Leftlist to Leftvector and Rightlist to Rightvector can be easily done using the list ranking algorithm. It is enough here to apply an algorithm ranking the list with m elements in $\log_2 m$ time with m processors on EREW PRAM (in our case $m = n/\log_2 n$). The sophisticated optimal list ranking algorithm is not necessary here. The weight of the segment is the number of brackets it contains. We can define the rank of the segment on a given list to be the weight of all segments to the beginning of the list (ending with this segment but excluding it). For example the rank of the segment [30..32] in Leftlist is 4. Hence its brackets are placed on the positions 5, 6, 7. Therefore Leftvector[5] = 30, Leftvector[6] = 31, Leftvector[7] = 32.

2.3. An alternative preprocessing

We give an alternative algorithm for the computation of splitting pairs. We compute the tree of reduced sequences of brackets (without positions) as in Fig. 4. It can be done using a balanced binary tree method. The operation of reducing the concatenation of two such reduced sequences can be easily seen to be associative. Using the tree we compute now the splitting brackets. A processor is assigned to each internal node of the tree.

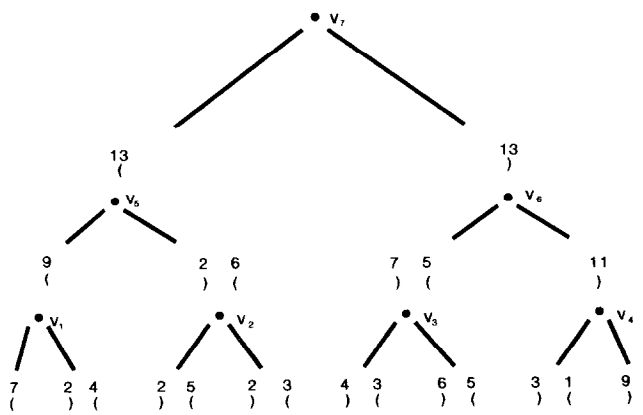

Fig. 4.

Consider the node v_5 in Fig. 4. The corresponding (to v_5) processor looks at the sons which contain $(^9$ and $)^2(^6$. The splitting left bracket is the 8th bracket in $(^9$, the value of v_1 in Fig. 4. Now the processor goes one step down and looks at the sons of v_1. They contain $(^7$ and $)^2(^4$. The 8th bracket at v_1 is the second left bracket in the right son of v_1. In this way one processor goes top down and computes the left

splitting bracket. The processors work in parallel. There are no write and read conflicts. At the beginning on each path from the root to a leaf we have $\log_2 n$ processors. In every step they are accessing data in distinct nodes. After the left splitting brackets are computed we compute right splitting brackets. Observe that certain nodes can have no splitting brackets and tobematched(v) can contain two empty lists for some v. This results in empty segments in Leftlist, Rightlist. However this causes no problems because we have still only $O(n/\log_2 n)$ segments in each list. For each pair of splitting brackets $(_i,)_j$ we perform in parallel MATCH[i] = j, MATCH[j] = i.

Once the matching function is computed the tree corresponding to the sequence of brackets can be computed without write and without read conflicts applying the method from [2]. We have proved.

Theorem 2.1. (a) *The matching function for a sequence of brackets can be computed by an optimal parallel algorithm on* EREW PRAM.

(b) *The transformation of the sequence of brackets to the tree can be done by an optimal parallel algorithm on* EREW PRAM.

We define the following "easy sorting" problem: Given a sequence of keys $x_1, x_2, \ldots, x_n$ whose values are integers such that $|\text{val}(x_i) - \text{val}(x_{i+1})| \leq c$, where c is a constant. Sort the sequence in a stable way.

Corollary 2.2. (a) *The easy sorting problem can be computed on* EREW PRAM *in* $\log_2 n$ *time using* $n/\log_2 n$ *processors.*

(*b*) *The bfs numbering of a tree can be done in* $\log_2 n$ *time using* $n/\log_2 n$ *processors.*

Proof. W.l.o.g. we can assume that all $x_i > 0$ and by inserting some new keys

- $x_i = 1$, $x_n = 1$,
- $|x_i - x_{i+1}| \leq 1$ for each $i = 1, 2, \ldots, n-1$,
- $x_i = x_{i+1}$ iff $x_{i-1} < x_i > x_{i+2}$ or $x_{i-1} > x_i < x_{i+2}$ for each $i = 2, \ldots, n-2$.

Then we assign a left bracket to the first element and a right bracket to the last one. To each other element x_i, $i = 2, \ldots, n-1$ we assign

- a left bracket if $x_{i-1} \leq x_i \leq x_{i+1}$,
- a right bracket if $x_{i-1} \geq x_i \geq x_{i+1}$.

Compute the function MATCH for the obtained sequence of brackets. If i is a left bracket then let $\text{NEXT}(x_i) = x_j$, where $j = \text{MATCH}(i)$.

Replace each left bracket by a right one and each right by a left one. Additionally insert some number of "new" left at the beginning and some number of "new" right brackets at the end to make the sequence well-formed. Compute again MATCH. Again for each left old bracket i with $\text{MATCH}(i) = j$ set $\text{NEXT}(x_i) = x_j$. In this moment the table NEXT gives the set of lists, each list contains our initial elements with the same value. Now it is easy to concatenate all lists in $\log_2 n$ time with $n/\log_2 n$ processors on EREW PRAM. The first one is the list of elements with

value 1, then the list of elements with value 2 etc. If one wants to transform the output list in a table then an optimal parallel ranking can be used.

(b) One can easily compute the dfs sequence of the tree and the level (distance from the root) of each node. Assume that the level is the value of a node (the nodes are keys). Now the stable sorting of the dfs sequence gives the bfs sequence. The assumptions of the easy sorting are satisfied. This completes the proof. □

Optimal algorithms on EREW PRAM for several other problems are consequences of the theorem. The only part where read conflicts were previously necessary in algorithms for these problems is the transformation of bracket sequences to trees.

Corollary 2.3. *The following problems can be computed by an optimal parallel algorithm on* EREW PRAM:

(a) *given a text of an arithmetic expression compute the value of the expression*:

(b) *given a text of a regular expression compute the nondeterministic finite automaton corresponding to this expression*;

(c) *recognition of input driven languages*;

(d) *parsing bracket languages.*

Sometimes instead of transforming sequences of brackets (representing structures of expressions) to expression trees the reverse transformation is useful. In all cases the table MATCH is crucial for the efficiency of parallel algorithms constructed. An example of such a transformation (trees to sequences of brackets) was given in Corollary 2.2. We point at two other such applications of sequences of brackets in parallel computations. The first one is related to the compressing of a tree to a tree with $n/\log_2 n$ nodes by contracting $n/\log_2 n$ chains in parallel on EREW PRAM, see [8]. The second is related to the isomorphism testing of trees on PRAMs. Again the tree is transformed to a sequence of brackets, see [4].

References

[1] K. Abrahamson, N. Dadoun, D.G. Kirckpatrick and T. Przytycka, A simple parallel tree contraction algorithm, Techn. Report 97-30, August 1987, Dept. Computer Science, The University of British Columbia, Vancouver.

[2] I. Bar-On and U. Vishkin, Optimal parallel generation of the computation tree form, ACM Trans. Programming Languages Systems **7** (2) (1985) 348-357.

[3] R.P. Brent, The parallel evaluation of general arithmetic expressions, *J. ACM* **21** (2) (1974) 201-208.

[4] B. Chlebus, K. Diks and T. Radzik, Testing isomorphism of outerplanar graphs in parallel, in: *Proc. MFCS'88*, Lecture Notes in Computer Science, Vol. 324 (Springer, Berlin, 1988) 220-230.

[5] R. Cole and U. Vishkin, The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time, *Algorithmica* **3** (1988) 329-346.

[6] R. Cole and U. Vishkin, Optimal parallel algorithms for expression tree evaluation and list ranking, in: *Proc. 3rd AWOC*, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 91-100.

[7] A. Gibbons and W. Rytter, An optimal parallel algorithm for dynamic expression evaluation and its applications, *Inform. and Comput.* **81** (1989) 32-45.

[8] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms* (Cambridge University Press, Cambridge, 1988).
[9] S. Kosaraju and A. Delcher, Optimal parallel evaluation of tree structured computations by raking, in: *Proc. 3rd AWOC, Lecture Notes in Computer Science, Vol.* 319 (Springer, Berlin, 1988) 101-110.
[10] G.L. Miller and J. Reif, Parallel tree contraction and its application, in: *Proc. FOCS* (1985) 478-489.
[11] W. Rytter and R. Giancarlo, Optimal parallel parsing of bracket languages, *Theoret. Comput. Sci.*, **53** (1987) 295-306.
[12] W. Rytter, A note on parallel transformations of regular expressions to nondeterministic finite automate, in: *Proc. Int. Workshop on Parallel Algotihms and Architectures*, Mathematical Research, Vol. 38 (Akademie Verlag, Berlin, 1987) 138-145.